L1 SPI – Algorithmic

S03 – The concept of script

Cyril Desjouy

15 juillet 2024

Preamble

Parts 1 to 3 of this practical present the concept of script and module. These parts will be done with your instructor and will form the basis of the course at which one to refer to in order to understand and deepen these notions. Part 4 is a practical application of the concepts seen in Parts 1 to 3 previous.

1 Introduction

Now that you have covered most of the basics of Python, it is necessary to move up a gear and start coding real ones programs. When it comes to writing a few lines of code, Jupyter is perfectly adapted. However, when it comes to coding a complete application, extending over several tens (see hundreds or thousands) of lines often distributed in several files, developers use generally an IDE (Integrated Development Environment). Spyder is one of them. among others, and this is the one we will use for our developments in the coming years. It has the advantage of being simple, effective, adapted to scientific computation and is also included in the Python distribution *Anaconda*.

— Start by launching Spyder

The Spyder interface is subdivided into several main sub-windows :

— a sub-window containing a text editor,

— a sub-window grouping Python and IPython consoles,

— a sub-window containing (among other things) an object inspector and a variable explorer.

— Start by typing an instruction in the text editor (for example `a=1`)

— Save this file [a] and execute it (F5 shortcut, and choose "run in the active Python or IPython console")

— Observe the information displayed in the console and the changes in the object explorer

a. Python scripts have the extension `.py`

Congratulations ! Even if it's not the classic "Hello World", you created your first **script** !

2 Interpretation of source files

2.1 Scripts

When running a program, the Python interpreter reads a source file containing code (a script) and interprets each instruction it contains. The first line of the file is read and interpreted first, the second is read and interpreted second, and so on. It is therefore important to check that on each line, the interpreter Python has all the necessary information to proceed with its interpretation !

1. Create a new script that you will call `FirstScript.py` in which you :
   — will define a function `myfct()` as follows :

```
def myfct(x):
    print("The square of {} is {}".format(x, x**2)
```

   — will use this function to display the square of the variable `a = 5` that you will have previously defined
2. Run the script and observe the result in the Python console (or IPython)

A script finally allows to group a set of instructions in the same one file. As mentioned before, the execution of a script consists of the interpretation of each line of the file successively.

2.2 Modules and *namespace* concept

1. Create a new script that you will call `SecondScript.py` in which you :
   — will import `FirstScript` as a module using the *built in* function `import`,
   — will define a function `myfct()` as follows :

```
def myfct(x):
    print("The cube of {} is {}".format(x, x**3)
```

   — will use this function to display the cube of the variable `a = 3` that you will have previously defined.
2. Run the script and observe the result in the Python console (or IPython)

Aren't there some unexpected lines ? Indeed, when the instruction `import` is used [1], the Python interpreter reads and executes the instructions contained in the imported source file. All lines of code interacting with the console are therefore interpreted conventionally and their results are displayed on the standard output. The `import` command creates a new object whose name is here `FirstScript` available in the *namespace* of `SecondScript` and that contains its own objects. This object is an object of type `module`.

The notion of *namespace* is a fundamental notion under Python. A namespace. is simply a container of names. It is used to allow the distinction between two elements with the same name. Let's consider the following example :

```
In  : import numpy          # We import numpy in the current namespace
In  : numpy.abs             # numpy provides the abs function
Out : <ufunc 'absolute'>
In  : abs                   # that is also defined in the current namespace ...
Out : <function abs>        # ... but is different from numpy.abs
```

In this example, we have access to two different objects with the same name :

— the function `abs()` of the module `numpy` whose name is prefixed by the *namespace* called *numpy*,

— the function `abs()` whose name is not prefixed and which is therefore in the current space.

The module name is thus used as *namespace* which allows not to overwrite objects already defined in the current *namespace*. For this reason, it is strongly recommended to avoid to copy directly a reference from one namespace to another unless to know exactly what you're doing :

```
In  : from numpy import abs   # We copy abs in the namespace current
In  : abs
Out : <ufunc 'absolute'>      # So we are overwritting the built in function abs
```

Each namespace is completely isolated from the others and you can check this by resuming your experiments on your files `FirstScript.py` and `SecondScript.py` :

1. For the import to work correctly, the Python interpreter must know the location of the file to be imported. If the file from which the import is performed is in the same directory as the file to import, the import will go right. If it's not the case and Python can't find the file to import in the `PYTHON_PATH` (this is the list of folders in which Python searches for modules, we will come back to this later), the interpreter will raise the `ImportError` exception.

3. Modify `SecondScript.py` as follows :
   - use the function `print` to display the value of the attribute `a` of the module `FirstScript`,
   - use the function `myfct` of the module `FirstScript` with as input argument the variable `a` in the current *namespace*. then the variable `a` in the *namespace* of `FirstScript`,
   - do the same thing with the function `myfct` of the current *namespace*.
4. Run `SecondScript.py`, observe the result in the console and conclude.

2.3 The special variable __name__

When the Python interpreter reads a source file (either directly in the course of its execution or indirectly in the course of its importation), it creates several special variables. One of them is of particular interest : this is the variable `__name___`.

1. Add instructions :
   - `print("Read FirstScript.py :", __name__)` at the end of the file `FirstScript.py`
   - `print("Read SecondScript.py :", __name__)` at the end of the file `SecondScript.py`
2. Run `FirstScript.py` and observe the value of the referenced object by the variable `__name__`.
3. Run `SecondScript.py` and observe the value of the objects referenced by the variables `__name__` in the different namespaces.

In the case of running `FirstScript.py`, the Python interpreter runs your script as the main program. You should see that the object `__name___` is a string containing the value "___main___".

In the case of running `SecondScript.py`, the Python interpreter also runs your script as the main program. You should see that the object `__name__` in the *namespace* of `SecondScript` is a string of characters that also contains the value "___main____". You should also see the value of the object `__main__` in the *namespace* of `FirstScript` which is a string containing the value "FirstScript". You will have found that in this case, the special variable `__name__` in the *namespace* of `FirstScript.py` no longer refers to the same object as when you have previously run the program `FirstScript.py`.

Conclusions :

- When a `.py` file is executed, the special variable `__name__` contains "___main___"
- When a file `MyModule.py` is imported from a file `MyScript.py` and that it is executed, the special variable `__name___` contains
  - "___main____" in the *namespace* of `MyScript.py`
  - "MyModule" in the *namespace* of `MyModule.py`

3 Python scripts : Good practices

The variable `__name___` is of paramount importance under Python since it facilitates and promotes code reuse. As you have seen previously, importing a source file results in the interpretation of all the lines of this file, which can be a problem when you want only access the functions defined therein and not the body of the program. That's when the variable `___name__` plays an essential role since it allows the definition of a code block that will be interpreted **only when the program is executed** thanks to the instruction :

```
if __name__ == "__main__":
    instructions
```

This instruction allows you to execute the script in its entirety if it is read as as the main program and to execute only the part before the instruction `if __name__ == "__main__"` if it is subject to a import.

The following Fig. **??** presents a classic script *template* that you can reuse at will for your future developments.

script.py

```
#!/usr/bin/env python                -> needed for direct execution
# -*- coding: utf-8 -*-              -> If missing, ASCII by default

'''
  doc of the program...              -> can span
  ... several lines
'''

import mymodule1 as myshortcut1      -> Modules used in program
...
import mymoduleN as myshortcutN

def mygreatfct(a):                   -> User function
    '''function docstring'''
    return a

...

if __name__ == "__main__":           -> Read the following only if script is executed !
    first_line = "Yeah! It begins"   -> begining of main program
    ...
    ...
    last_line = "Yeah! It ends"      -> end of main program
```

FIGURE 1 – Recommended frame for writing Python scripts

By using this type of structure, you ensure that each code can be reused simply in other applications by importing your scripts as a module in order to access the functions that you will have previously developed.

The next step is to test this *template* in the case of files you have previously created :

1. Add the condition on the variable `__name__` at the right location in your file `FirstScript`
2. Run `FirstScript.py` and observe the result in the console
3. Run `SecondScript.py` and observe the result in the console

4 Application : Creating a module for electrokinetics

The aim here is to repeat the developments you have carried out during the `S09E02` of semester 1 to create a module offering tools for creating electronic filters of the 1er and 2nd order.

1. Create a file `eec.py` that will be a module containing tools for electrokinetics and which will contain :
   - A class `Filter` that can be instantiated with :
     - a mandatory argument `freq` representing the frequency axis on which one to calculate the filter,
     - a mandatory argument `wc` representing the filter cut-off pulsation,
     - an optional argument `Q` representing the quality factor of the filter (second order filters only),
     - an argument of the type `string` specifying the type of filter (high pass, low pass, band pass),
     - an argument of the type `int` specifying the order of the filter (order 1 or 2),

     and providing the methods :
     - `order1filter()` returning the transfer function of an order filter 1 (RC or RL filter),
     - `order2filter()` returning the transfer function of an order filter 2 (RLC filter).
   - A function `modphase()` taking as input arguments a transfer function and returning its module (**en dB**) and its argument (**en radians**).
   - A function `bode()` taking as input arguments a transfer function, the frequency axis and a string representing the title of the plot. This function will use `modphase()` to trace the Bode diagram of the filter (amplitude response and in phase on two sub-figures arranged on the same line). The figure **??** shows an example of a plot produced by the function `bode()`
2. Add in the file `eec.py` a series of tests allowing to validate the functions and classes you have defined. These tests do not will not be executed when using `eec.py` as module. These tests will include, for example, calculating and tracing transfer functions :
   - of a first-order low-pass filter
   - of a second order bandpass filter
   - of the multiplication of two second order filters
3. Create a file `test_filters.py` in which you will import your module `eec` and use it to create a series of 10 filters low pass whose cut-off frequencies will be linearly distributed between 100 and 500 Hz. Overlay the plots of the transfer functions of these 10 filters.

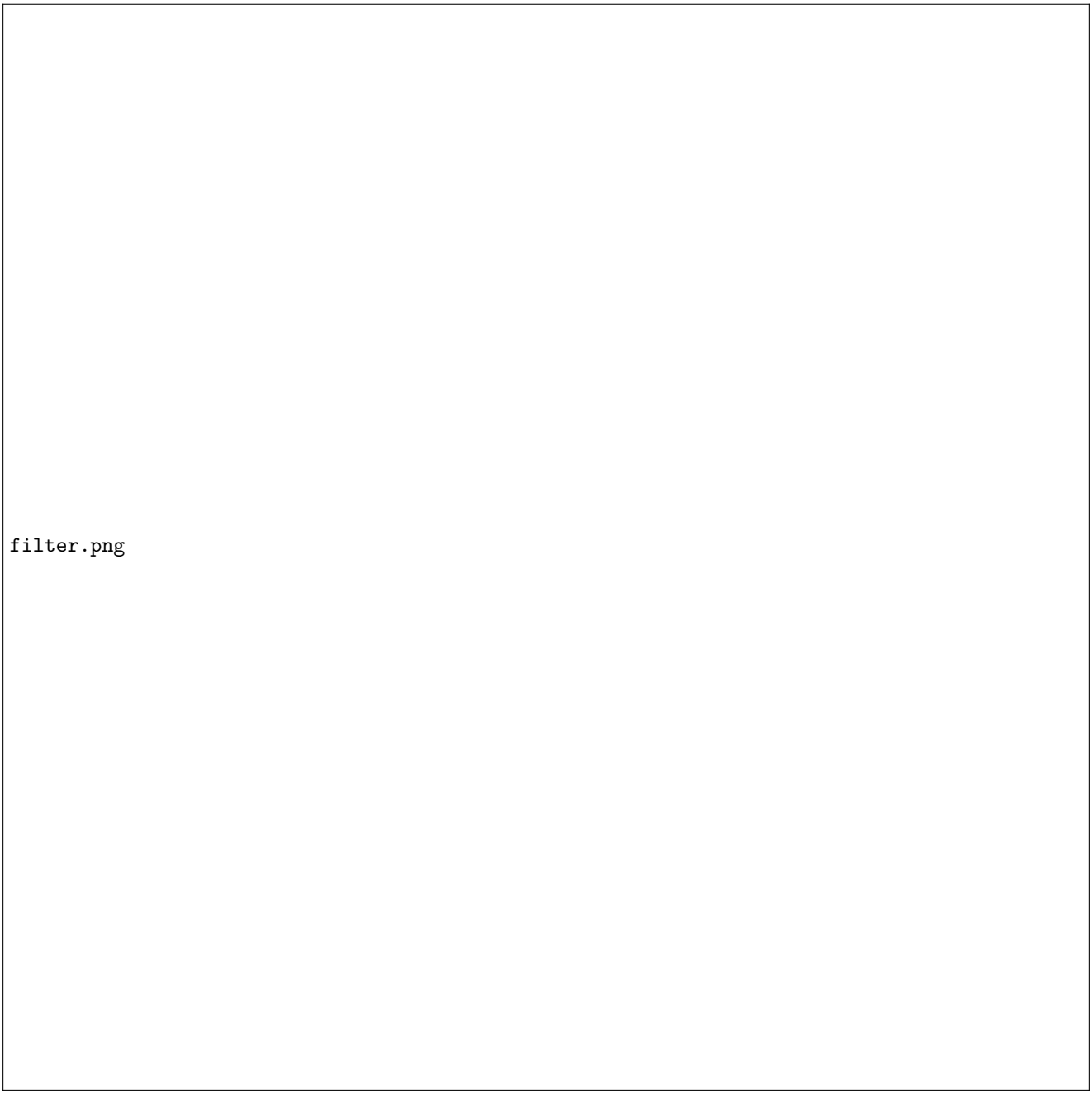

FIGURE 2 – Example of a path produced by the function `bode()`

.